

Communication Abstractions in Concurrent Processing

Waldemar Horwat
waldemar@acm.org

Abstract

Parallel and distributed programs are entering the mainstream—a 1024-processor Connection Machine is now the fastest computer in the world, while at other places users are routinely running distributed programs on networks of workstations. One of the most important aspects of parallel programming is the abstraction of the communication channels between processors as presented to the programmer—how does one implement communication among processes in a program in such a way that is very efficient both when the processes are on the same node and when they are on remote nodes?

This paper presents and compares several approaches of viewing communication, including cache-coherent shared memory, message passing, and higher-level protocols. The important hardware constraints will be examined, and some predictions for the future will be presented. A number of literature references are provided.

1. Introduction

Issue

As the densities of integrated circuits increase over time, parallel and distributed processing are becoming more important and widespread. Unfortunately, unlike in sequential computing, there is a lack of consensus on the mechanisms needed for parallel and distributed computer systems. In particular, communication primitives differ widely among parallel architectures and operating systems. Some parallel programs communicate by sharing data in memory, others by sending messages such as AppleEvents, and still others by higher-level operating system calls.

Why should Macintosh developers concern themselves with parallel programming, and, specifically, communication in parallel programming? There are a couple of reasons: First, networks are becoming very widespread and offer a great resource for running large or multi-user applications. Second, the densities of integrated circuits continue to increase geometrically [25], and by the turn of the century will permit densities of 100 million transistors on a chip (by comparison, a 68040 has slightly over one million transistors). Increasing cache sizes and sizes and numbers of logic units in processors will shortly reach the point of diminishing returns, forcing architects to put multiple processors onto a single chip in order to significantly improve performance for the coming generation of computer applications. The goal of this paper is to explore communication paradigms that let programs run efficiently on a wide range of parallel architectures, from fast sequential processors to tightly-coupled parallel systems to distributed systems.

Outline

This paper examines and contrasts three communication paradigms: cache-coherent shared-memory, message passing, and high-level distributed system communication services. Even lower and higher-level abstractions such as electronic mail handling exist, but they will not be discussed here.

One of the predictions made here is that as technology advances, the shared memory and message passing paradigms are going to merge into what could be a universal localized communication paradigm. However, at this point it's not clear whether distributed communication services can be neatly integrated with the other two.

The three communication paradigms are discussed individually in Sections 2, 3, and 4. Section 5 compares them and describes the coming unification of shared memory and message passing.

Copyright © 1993 Waldemar Horwat

2. Shared-Memory Architectures

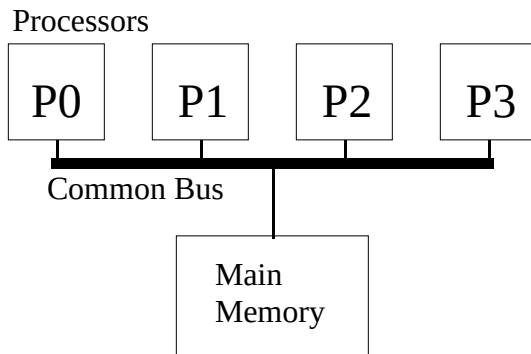


Figure 1. Shared-Memory Architecture

A shared-memory architecture appears to the programmer as several processors sharing a common pool of memory as in Figure 1. Each processor can read and write words of memory; a word of memory typically appears at the same address for each processor.

Caching

The architecture in Figure 1 suffers from the absence of caching. A few architectures without caching have been proposed; Horizon [33] is an interesting example in that it demonstrates how task switching can be used to hide the latencies of memory operations to the extent that the processors can keep themselves busy without caches—each processor can work on 1024 tasks at the same time! The bus has been replaced by a three-dimensional mesh network which has the throughput to handle all of the memory requests. Nevertheless, this architecture is not resource-efficient, as it has N^3 memories and only N^2 processors, for $N=16$.

The vast majority of architectures under consideration today make use of local caches attached to processors as in Figure 2 (Some architectures are composed entirely of caches [24][32]). The data in these caches has to be kept coherent in some manner so a processor will not read old data from its cache after another processor has modified the data. Note that a shared-memory architecture does not have to be implemented in hardware according to the block diagram in Figure 1; in fact, some of the architectures discussed below have different topologies, don't use buses at all, or communicate by message passing.

Two important aspects of shared-memory systems are support for synchronization and the ordering semantics. Synchronization in its simplest form can be achieved using indivisible test-and-set operations; other variants exist which are fundamentally more powerful. The simplest shared-memory systems have strong consistency ordering semantics, which means that there exists the concept of a global time such that processors read and write words one at

a time, and events are perceived in the same order by all processors. Other consistency semantics exist with weaker assumptions somewhat reminiscent of Einstein's special relativity; one of these will be discussed later in this section.

Bus-Based Shared-Memory Architectures

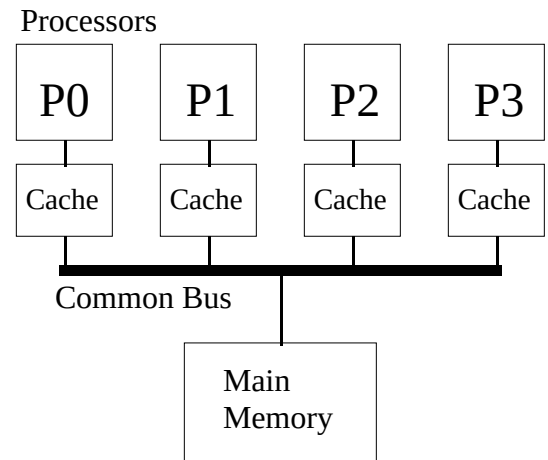


Figure 2. Shared-Memory with Caches

The processors communicate only with their caches, which then communicate with the main memory and other caches through a common bus.

If one were to connect several processors directly to main memory as in Figure 1, the performance would not be much better than having one processor because the memory bus would saturate. Putting caches between processors and the memory requires a protocol to ensure that stale data does not remain in caches. Goodman's protocol [20] utilizing snoopy caches is one of the simplest and most popular such protocols. Versions of this protocol are hardwired in many modern microprocessors for building small concurrent systems. The PowerPC 601 implements MESI, a slight variant of this protocol [10].

Goodman's Protocol

In Goodman's protocol, processors operate on cache lines (typically 16-128 bytes on today's architectures). Every main memory access either reads or writes a cache line; if a processor wants to write a single word, it has to read the entire line into the local cache, alter the word, and later write the line back to main memory. Each processor's local cache line can be in one of four states with respect to a particular memory address a :

- **Invalid (I):** There is no valid data cached here or the cached data corresponds to an address other than a .
- **Shared (S):** Data from address a is cached here and the cached data corresponds to main memory contents.
- **Exclusive (X):** Data from address a is cached here, the cached data corresponds to main memory contents, and no other processor can have a cached copy of this line.
- **Dirty (D):** Data from address a is cached here, the cached data has been modified, and the change has not been transmitted to main memory yet.

Each local cache's actions on a read, write, and line flush are as follows:

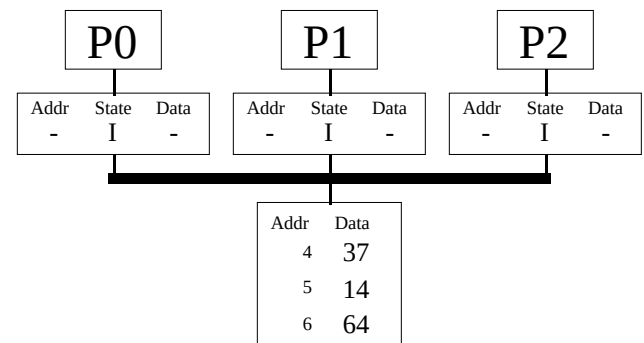
- **Local processor reads line:** If the line is shared, exclusive, or dirty, provide the data locally. If the line is invalid, request the line from main memory. At this point either the main memory will provide the data or another processor's cache will notice that it has a dirty copy of the line, in which case the latter processor will provide the data both to the main memory and to the original processor. This is called a snooping operation because when one cache requests a main memory line, all the others snoop the bus to check whether they have a dirty copy of that line.
- **Local processor writes line:** If the line is dirty, the write proceeds locally. If the line is exclusive, the write proceeds locally and the line state is changed to dirty. If the line is shared, the write is done both to the local cache and to main memory (thereby purging this line from all other caches; see below), and the line state is changed to exclusive. If the line is invalid, the line is first read from main memory¹ as above and then the cache proceeds as in the shared case.
- **Line pushed out of cache** (i.e. the cache line is needed to make room for a different memory address): If the line is dirty, write it back to main memory. Change the state to invalid.

In addition, each cache snoops all of the other caches' transactions with the main memory through the common bus and performs the following actions:

- **Remote processor reads line:** If the line is in the dirty state here, change the state of the line to shared and broadcast the data on the bus, telling the main memory to update its copy; the remote processor also gets the broadcasted data. If the line is in the exclusive state here, change the state to shared.
- **Remote processor writes line:** If the line is in the shared, exclusive, or dirty state here, invalidate it².

Example

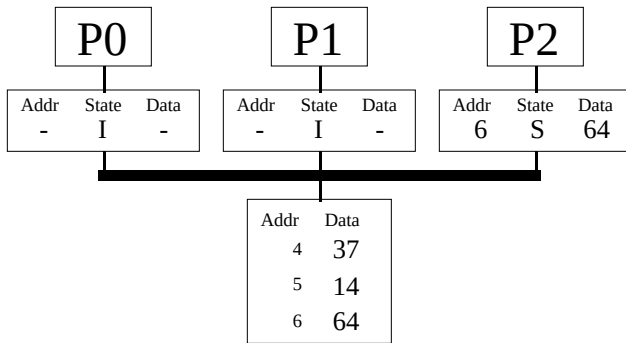
To illustrate the protocol, consider an example with three processors, each with a one-entry cache.



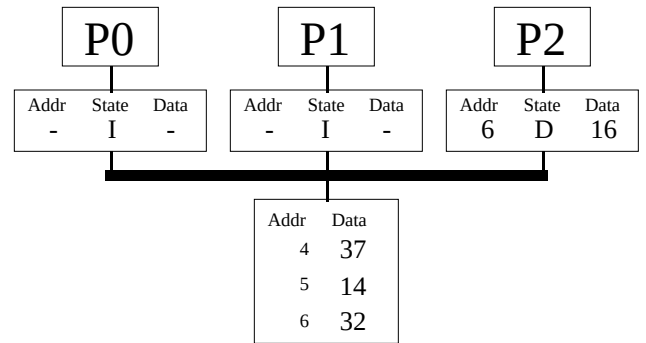
The processors start with empty caches. The main memory initially contains the value 37 in the line at address 4, 14 at address 5, and 64 at address 6 (the lines are typically 16-128 bytes long but are represented by single bytes for the sake of the example).

¹This read is only necessary if the processor modifies part of the line such as a single word. For this reason, some processor architectures, including the PowerPC, provide instructions to clear an entire line, while the 68040 provides an instruction to copy an entire line of data.

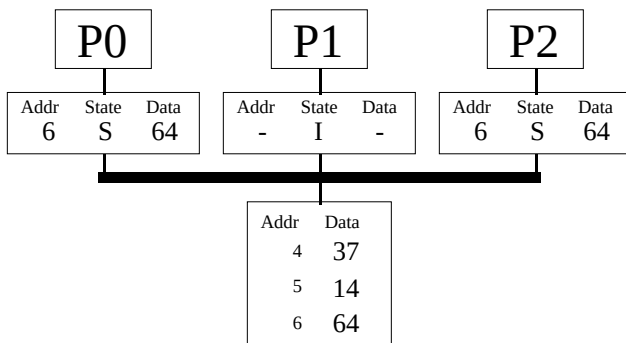
²Getting the data from the bus and putting it into the local cache might create a stale copy because the remote processor now places its line in the reserved state and will not send out notifications of further writes.



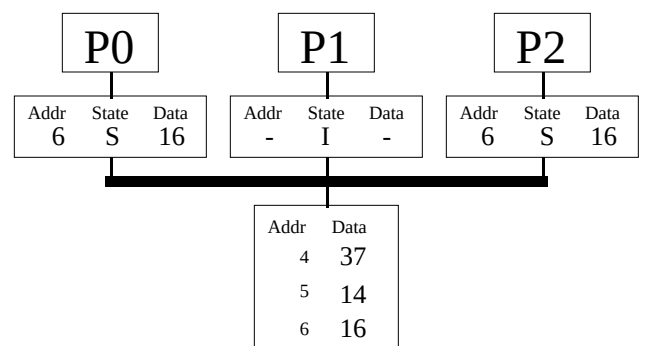
After processor 2 reads memory line 6, the data is placed in its cache in the shared state³.



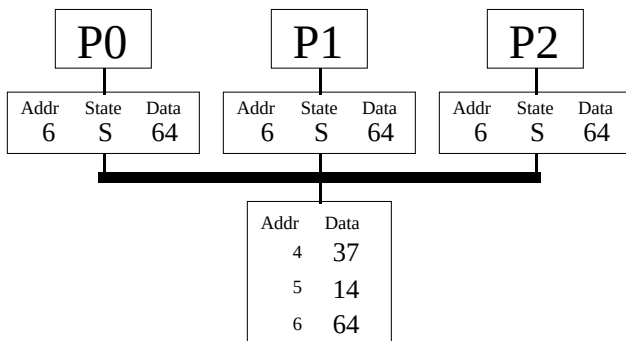
At this point processor 2's cache is in the exclusive state, so it can modify line 6 locally at will without writing through to main memory.



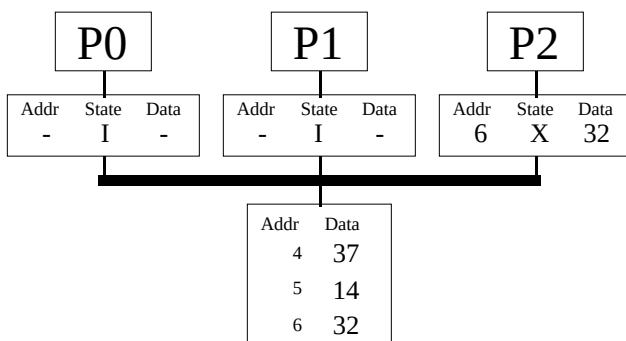
Processor 0 then reads memory line 6 with the same effect.



When processor 0 requests a read of line 6, processor 2 provides the data and updates main memory. Both caches are now in the shared state.

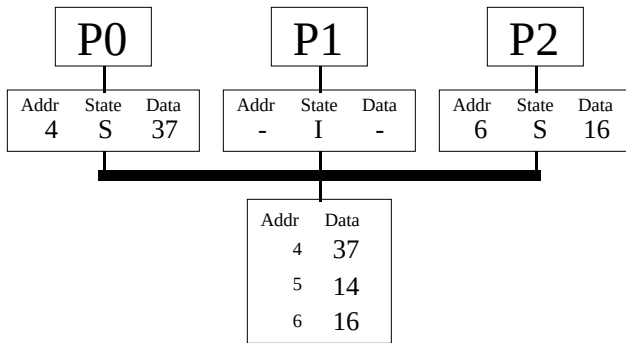


Processor 1 joins in and reads the same memory line.



Now processor 2 writes 32 to line 6, invalidating the other two processors' caches and writing through to main memory.

³The state is not reserved because P2 has no way of knowing that the line is not already cached somewhere else. An optimization of the protocol provides an additional bus signal to P2 to let it know about this.



When processor 0 reads line 4, it displaces its existing cache line. The previous line wasn't dirty, so no writeback is necessary.

Other Bus-Based Shared-Memory Architectures

Even with caches, the bus in the above scheme becomes a bottleneck for more than three or four processors. Furthermore, every cache has to get involved in every main memory transaction, which inherently bounds the performance of the system. The following alternative bus-based schemes try to overcome these problems.

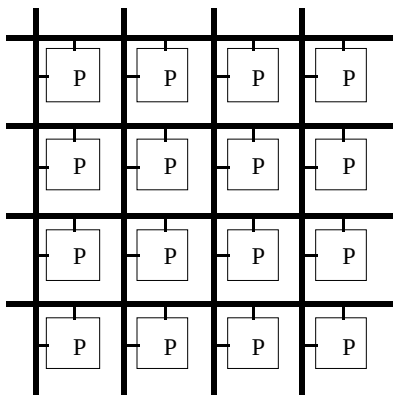


Figure 3. Bus-Grid System
Each processor resides at an intersection of two buses.

Bus-grid systems such as the Wisconsin Multicube [22] and Aquarius [9] rely on a grid of horizontal and vertical buses for communication, with processors present at the buses' intersections. The Wisconsin Multicube has memories attached to the vertical buses, while Aquarius has a portion of memory present with each processor.

The Wisconsin Multicube requires each processor on a column to maintain a table of the dirty lines in *all* processors on that column. Memory requests are broadcast on a row bus. If one of the processors on that row notices that the line is dirty in its column, it will identify itself and route the transaction to a processor in that column; otherwise, the processor at the intersection of the given row and the column corresponding to the memory containing the line will process the transaction. The protocol includes several more transactions to maintain each line in either shared or exclusive state. The main weakness of the protocol is that

every time a write is made to a line in the shared state to put it in the exclusive state, an invalidation request has to be broadcast to all processors, which again bounds the performance of the system.

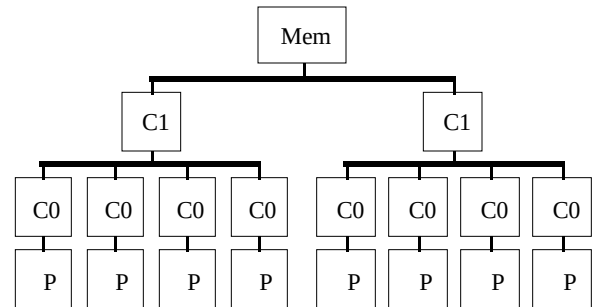


Figure 4. Hierarchical System
This system is a hierarchy of Goodman systems with several levels of caches between the processors and main memory.

Hierarchical systems such as a hierarchical version of Goodman's cache coherence protocol [48]. The computer is organized as a tree of buses. The main memory is attached to the top-level bus; the other pairs of levels are connected by caches. The processors at the leaves are connected to the lowest-level buses via local caches. An important invariant is that every cache entry must also be present in all of its parents' higher level caches. Thus, the root cache of a subtree can immediately determine that a particular line is not present in its subtree. An undesirable consequence of the invariant is that the higher-level caches must be very large. The Goodman protocol is followed at each level of the hierarchy, with the addition that when an existing entry in a higher-level cache is invalidated, it is also invalidated in all of the cache's children.

The Data Diffusion Machine [24] is also a hierarchy of buses, but with an interesting twist: it has no main memory—all data resides in caches. When a cache fills up, it either erases or moves its data somewhere else on the machine. Cache lines have no home—they are created on the nodes that allocate them and then diffuse through the machine. The Data Diffusion Machine protocol is complicated by a multitude of transitional states that manage reads, writes, locks, and cache spills in progress.

The VMP-MC paper [23] contains an analytical model and some simulation results to determine by how much the traffic in upper-level buses is reduced by sharing lower-level caches as compared to having an equivalent total amount of cache memory separated among the processors on a flat system. For realistic assumptions, the ratio is about 0.5, meaning that only half as many requests propagate to the upper-level buses as would if the hierarchy were flattened; thus, except in special cases, the hierarchical systems considered here fail in significantly reducing the bandwidth needed on the top-level bus.

Buses Can’t Keep Up

The above results look discouraging, so perhaps there is something wrong with the concept of a bus itself. Buses have long been popular in computers. They allow many devices to be connected and offer the advantage of cheap broadcasting, which allows various snoopy caching techniques.

Nevertheless, as technology advances, buses are becoming impractical. Due to their large dimensions and propagation time limitations, even high-performance buses have cycle times on the order of 20 to 30ns, while point-to-point links can be clocked at 2ns. Simulations indicate that 2ns point-to-point rings of 4 to 16 devices have approximately the same throughput as buses clocked at 4ns [40]. The ring’s latency can be slightly worse than a bus, but the throughput is much better. Since 4ns buses are not practical at present⁴, point-to-point networks are a better choice. Moreover, as the number of devices on a bus is increased, the cycle time increases, while a network’s cycle time is not affected.

The trend away from buses is clearly visible. Kendall Square Research’s KSR-1 [32] is an example of how rings of point-to-point links can replace buses. The KSR-1 is similar to the Data Diffusion Machine presented earlier. It consists of one or more fast unidirectional rings, each with 32 processors. Up to 33 rings can be connected together by a second-level ring. Like the Data Diffusion Machine, the KSR-1 consists entirely of cache—there is no main memory. Unfortunately, the KSR-1 has a hierarchical organization and may suffer from bandwidth problems on the top-level ring as with the VMP-MC.

Network-Based Shared-Memory Architectures

A fundamentally different approach is to allow an arbitrary communication network between the processors and memory and to rely on a protocol to notify processors that their local caches’ data is stale. Most of these protocols [42] rely on the concept of a directory in main memory, which keeps track which processors have copies of the data so a

writer can invalidate all outstanding copies of a cache line. These protocols can be implemented fully or partially in hardware, or they can be software-based. The merits of compiler-scheduled software cache coherence are studied in [38] and [1], concluding that this form of coherence performs well for structured problems but is highly sensitive to minor variations in less structured situations. Hardware, or, at least, hardware with software to handle difficult cases appears to be the best approach here.

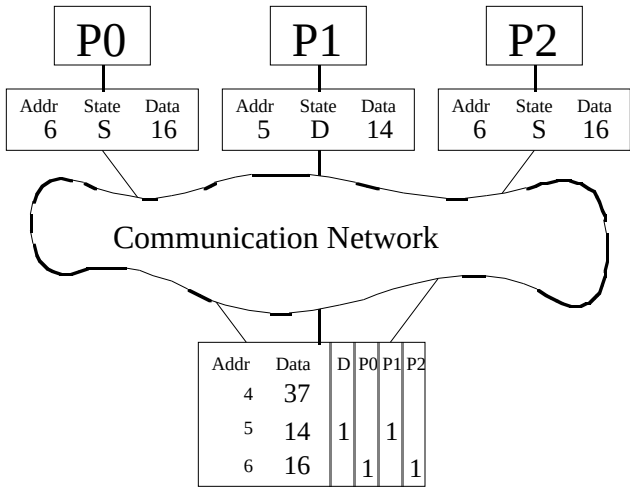


Figure 5. Bitmap Directory
The main memory directory keeps track of which processors have copies of data.

The original directory approach [11] is to maintain a bitmap directory (Figure 5) with each main memory line. One directory bit in each line corresponds to each cache in the system; the bit is set whenever the corresponding cache contains a copy of the data line. An additional D bit is set if the data is dirty in a cache (in which case exactly one other bit in the bitmap is set). The processor caches run a protocol similar to Goodman’s except that when a cache wishes to write to a line that isn’t already dirty, it looks up the processor numbers of other processors caching that line in the directory and then sends invalidate messages to

⁴Very high-speed buses exist in restricted applications such as the 500MHz RAMBus, which has a master at one end and slaves along the rest of the bus; different clocks are used for sending and receiving, and the entire bus has to be physically small.

them⁵. The size of the directory bitmap is equal to the number of processing nodes in the system, so this scheme is only suitable for small systems.

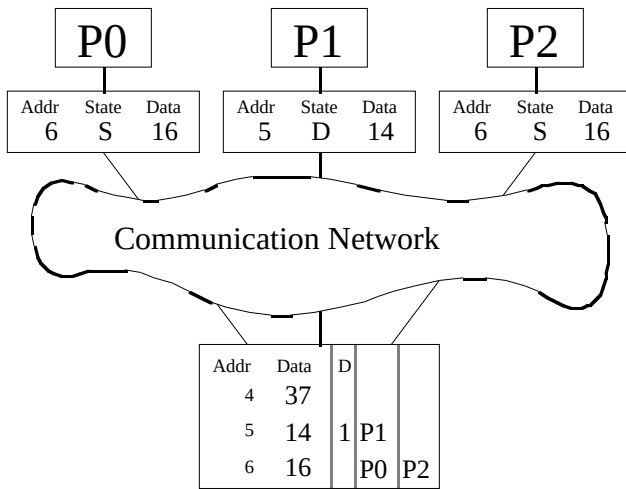


Figure 6. Limited Directory

The directory is smaller but can only keep track of a subset (at most two in this example) of shared copies of a data line.

To reduce the size of the directory, the directory format can be changed to an array of k pointers [3]. The pointers refer to at most k caches which can have shared copies of the line. If all k pointers are in use and another cache would like to get a copy of the data, one of two things happens depending on the implementation: one of the original k caches gets invalidated, or the directory line enters a state where the set of caches with copies of the data is unknown and a broadcast will be required to invalidate all caches the next time the line's data is modified. This scheme either causes thrashing in some cases or requires occasional broadcasts; however, statistics indicate that these situations are rare in most programs even for small values of k such as 2 [47]. Another promising approach is to escape to software when the number of shared copies exceeds the capacity of the directory [2].

An even better approach is the Scalable Coherent Interface (SCI) scheme [28], which has a directory with one pointer in the main memory. When multiple caches have copies of the data, the pointer points to the first cache of a doubly-linked list threaded through the caches. Specifically, when a cache A requests data from the main memory and the data is already cached in cache B , the main memory immediately responds with a message telling A to ask B instead (and changes its head pointer to point to cache A). Cache A then asks cache B to get the data and notifies B that A is now the head. If cache A were doing a write, it would ask B to invalidate itself and send back a pointer to B 's successor, if any; A would then proceed to purge all the other successors from the linked list in the same way. When a cache line is replaced, the cache can remove itself from the doubly-linked list by linking its neighbors together. This scheme requires four network transactions for a shared read and a number of

transactions and latency proportional to the number of copies being invalidated for a write. This scheme is an IEEE standard for very fast communication services for a network of nodes.

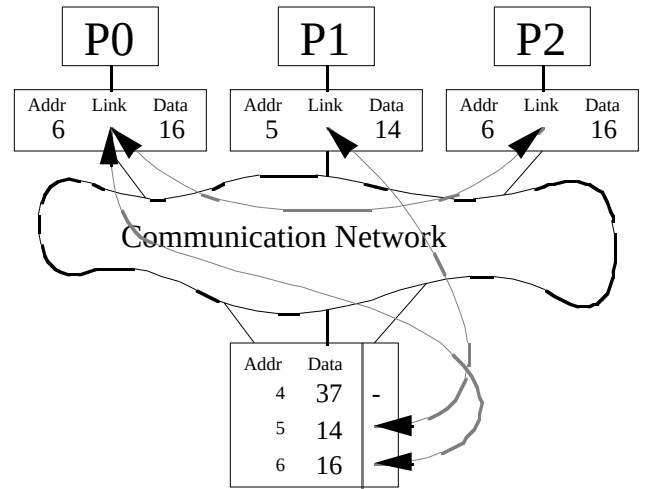


Figure 7. Linked Directory

The caches containing copies of data in memory are linked into a doubly-linked list, represented here by gray lines. The first cache in the list has the authority to invalidate the others if it wants to acquire exclusive ownership of a line or write to it.

A variant of this scheme is the Stanford Distributed-Directory Protocol (SDD) [43]. This protocol uses singly-linked lists of caches instead of doubly-linked ones; when a cache line is replaced, the cache must invalidate all caches downstream from it first. SDD includes an optimization in the common case where cache A reads a value that is already in cache B —the main memory forwards A 's request directly to B rather than replying back to A and having it contact B . This reduces the number of messages from four to three but might deadlock if the network is full, so the protocol uses the four-message scheme in that case. Writing and obtaining exclusive access to a cache line is done as in SCI.

⁵The actual protocol is a little more complicated to handle the case of two or more processors wanting to modify the same line at almost the same time.

Topology

So far the directory protocols were examined without regard to network topology. The two main kinds of network topologies are hierarchies or variants of a multicube.

Hierarchies allow recursive protocols and can map well to hierarchies of physical packaging—chips, boards, and cabinets—but the top levels can become bottlenecks. Both kinds of topologies can take advantage of locality. As observed before and in [5], trees or rings suffer from bandwidth problems at the root.

Meshes and multicubes tend to have better performance characteristics. Their bandwidth scales into large sizes much better, and machines with thousands of processors have been built. Current network speeds are fast enough that the exact mesh or multicube topology doesn't matter much, at least until one starts to build machines with tens of thousands of processors. On the J-Machine [14][15], one can ignore topological locality altogether without burdening the network until the machine size exceeds a thousand processors [26], and this is true for multicubes in general [29].

Other topologies are also possible, such as a fat tree⁶ in the Connection Machine 5 or the hybrid topology in Dash [34], which is a mesh of nodes, each of which has several processors on a local snooping bus. Hybrid topologies can match physical packaging I/O constraints well.

Does topology matter? For the foreseeable future, at least some programmers will try to take advantage of hardware topologies when mapping their applications to get the best performance. Some of the architectures, especially the hierarchical bus/ring-based ones, don't work well unless the mapping limits traffic across the root; others, such as meshes and fat trees, provide plenty of bandwidth for now. Explicit hardware mapping can run into problems if there are faults present in the machine, and it is non-portable. In the end, for most applications it's probably best to request a logical geometry and let the firmware pick an appropriate mapping. In return, the machine topology should be able to support the desirable mappings efficiently.

Strong and Weak Consistency

There are three important shared memory efficiency enhancements that require some knowledge of the semantics of the application. The first is the use of *weak consistency* [16] for most data accesses.

A parallel system is *strongly consistent* if all memory reads and writes can be placed in a total chronological order. Strong consistency is expensive to achieve in practice because it requires that if processor *B* reads a variable *x* written by

processor *A*, *B* then performs a write to *y* which is read by processor *C*, and finally *C* reads *x*, then *C* must get the new value of *x* produced by *A*. Thus, *A*'s write of *x* must reach *C* before any communication from *A* can reach *C* through *B*. In practice, this requires that each processor wait until a write is completed before proceeding, introducing a significant performance bottleneck.

A weakly consistent system requires only that the memory operations be serializable locally—a processor's memory operations appear chronologically consistent from its frame of reference but not necessarily from the other processors' view. Furthermore, a certain set of memory locations or operations⁷ can be distinguished as being *synchronizing*—synchronizing operations must be strongly ordered. Finally, synchronizing and non-synchronizing memory accesses must not be reordered or overlapped. Synchronizing operations are used for items such as locks and queues, while non-synchronizing operations are used for general data. A processor can continue running immediately after a non-synchronizing write is issued.

The PowerPC architecture supports weak consistency and provides the SYNC and EIEIO (!) instructions [10] as memory barriers.

Locks

The second architectural enhancement aims to improve performance when dealing with heavily contested locks. When *n* processors are all spinning, trying to acquire the same lock using test-and-test-and-set, $O(n^2)$ network operations are needed in unoptimized cache coherence schemes before each processor can get the lock once because the lock is written *n* times, each time invalidating $O(n)$ cached copies on the other processors.

⁶A fat tree is a tree in which there are many redundant links at the higher levels. These links provide greater bandwidth and fault tolerance.

⁷Two variants of read and write instructions can be provided. Some architectures provide a barrier instruction instead that waits until all pending memory operations complete before proceeding.

The solution is to extend the protocol to queue the processors waiting for a contested lock; when a processor releases the lock, it passes it to the next one in the queue, requiring only $O(n)$ network operations.

This approach is presented in [21], where three primitives are introduced: test-and-set, unset, and queue-on-synbbit. The last one notifies the hardware that this processor would like to be appended to a queue waiting for a lock, if there is one. The same operations can be used to efficiently implement other synchronization primitives such as fetch-and-add, work queues, and barriers.

Multitasking

Technology trends indicate that the speeds of processors will continue to rise much faster than memory or communication latency (but not bandwidth) for the foreseeable future. Whereas cache miss penalties were equivalent to one or two instructions a few years ago, a future processor will be able to execute tens or hundreds of instructions in the time it takes to fetch a word from main memory or another processor. Thus, it's important to avoid stalls on cache misses.

Memory latencies can be hidden (for the time being) on sequential processors by executing other instructions until the result is needed. The same thing can be done on parallel computers as long as excess parallelism is available—a processor can switch to another task while the original one is waiting for a remote reply. Thus, processors will evolve to be able to run more than one instruction stream at a time; this trend will be examined further in the following sections.

Why Does This Matter?

The details of the implementation of shared memory cannot be completely hidden from the programmer. The weak consistency optimization must be exposed, as it affects the semantics of the program, even if one is programming in a high-level language like C. Optimizing locks affects performance only, but it is also exposed through the communication abstraction to let the programmer notify the hardware about lock operations; furthermore, practical hardware may have limitations such as only supporting one lock per cache line. Multithreading will be a revolutionary change in the design of processors, and it is inevitable.

3. Message-Passing Architectures

The second paradigm for communication within a parallel application is passing messages. A message-passing system is conceptually organized as in Figure 8: each processor has a local memory and access to a network which routes messages between the processors.

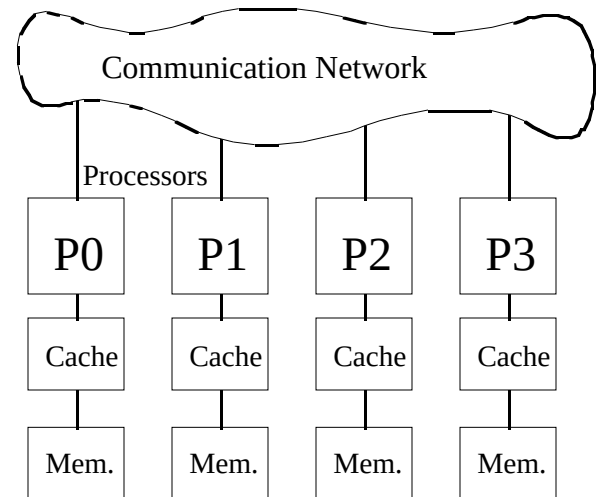


Figure 8. Message-Passing Architecture

The processors communicate with other processors via a network. Each processor's memory and cache are local, so there are no coherency problems.

Although message-passing architectures have existed for a while, they typically provided only high-overhead communication facilities—sending a message involved an operating system call and a protection domain switch, which by themselves consumed about a millisecond. This may be acceptable if the communication medium is an Ethernet, but not in a modern parallel computer. Decreasing communication overheads can improve program running times by more than an order of magnitude for a variety of applications [5]. The ability to *efficiently* pass messages between processors and have the destination processor perform some activity upon receipt of a message can be used to implement a great variety of programming language constructs such as procedure calls and returns, object migration, and barriers.

Achieving low communication overhead requires streamlining of the way messages are sent and received. The precise mechanism used to send and receive messages can vary, but there are some things the hardware should provide: efficient

sending of messages from user code, queueing of incoming messages, and atomicity. Two projects provide good illustrations of possible implementations of efficient message passing: Active Messages and the J-Machine.

Active Messages

Active Messages [17] is a simple communication abstraction for parallel message-passing architectures, including existing systems; it is currently implemented on the CM-5, nCUBE/2 and the J-Machine. A compiler can use Active Messages as a basic mechanism to implement remote procedure calls, block sends, and other communications. The emphasis is on reducing the overhead of sending and receiving messages—conventional libraries use hundreds or thousands of microseconds to compose and dispatch messages, while active messages can be composed and dispatched in a few microseconds on the same hardware.

An active message contains a program counter value and zero or more words of data. When such a message is received by the destination processor, it begins executing a message handler starting from the supplied PC value. That handler then interprets the rest of the message and decides what action, if any, to take. The source processor is required to know the address of the handler on the destination processor; this generally does not present a problem if the handlers are fixed in each processor's address space.

An active message handler must be short and non-blocking to avoid network backups and deadlocks. Thus, active message handlers themselves are unsuitable for performing nontrivial computations; they cannot, for example, send messages at the same or lower priority level as the received message. Instead, an active message handler can be used to receive arguments for a procedure call and schedule the procedure invocation itself as a background process. A program executes a remote function call by sending the address of the procedure, arguments, and a return pointer in a message to a remote node; the message's PC points to a handler that queues the procedure invocation in a local task queue, which is later retrieved by a background dispatcher. When the procedure finishes, it sends the result back in another message.

Active Messages can be implemented reasonably well on existing hardware, but a few simple hardware optimizations can make this model more efficient. It helps to be able to compose and send messages directly from registers without having to store them in memory and use DMA. The entire received message should be directly accessible so it does not have to be read one word at a time or copied into memory. The details of the routing hardware should be hidden from the program so it can just designate the destination address and message contents in order to send a message. Finally, message sending and receiving should be done at the user level for most messages to avoid costly kernel calls.

J-Machine

The J-Machine [14] has `SEND` instructions to inject message words directly into the network, making it very easy to inject messages. On the receiving end, the J-Machine queues messages in a circular buffer on each node (this is an extremely fast operation due to the memory design). The J-Machine processors automatically handle incoming messages one at a time, dispatching on the PC values in their first words. If a J-Machine procedure calls another procedure, it executes the `SUSPEND` instruction to let other messages run; the compiler ensures that the procedure copied all of its needed arguments from the message into an activation frame by this point. Figures 9 and 10 show the J-Machine procedure call and return messages.

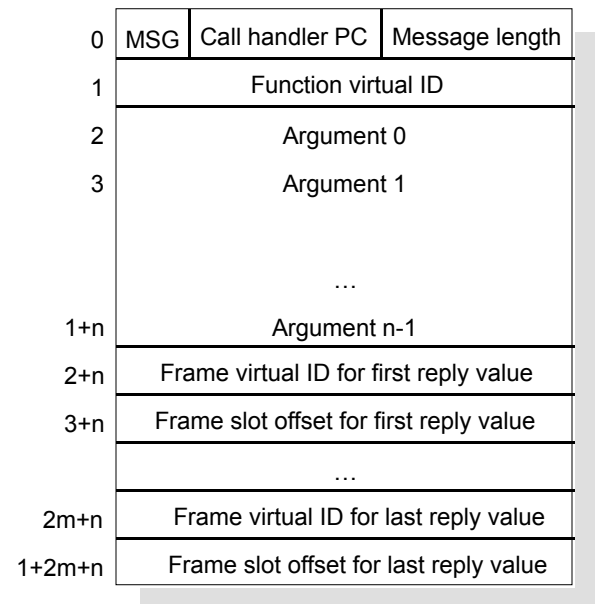


Figure 9. J-Machine Call Message
This message is used to initiate a remote procedure call. The Call handler PC is the address of a four-instruction routine that translates the function virtual ID and then jumps to the function code.

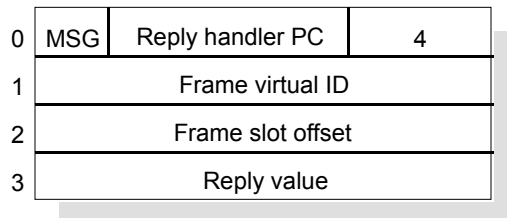


Figure 10. J-Machine Reply Message
This message returns the result of a remote procedure call.

Unlike the Active Messages proposal, a J-Machine message handler can perform arbitrary tasks, including sending other messages. Procedure calls are executed directly rather than being stored in a task queue. Unfortunately, this means that the hardware message queue frequently overflows if the procedure calls are time-consuming; messages in the hardware queue then have to be copied to other portions of memory. The Active Messages architecture, on the other hand, avoids the problem of message queue overflows, but its task queue is subject to overflows, and, furthermore, every procedure invocation must be copied rather than only non-leaf procedures as on the J-Machine. Quick requests do get handled faster under Active Messages, but the same thing could be done on the J-Machine by using more priority levels⁸.

The J-Machine also provides for hardware support of a user-defined translation between virtual and physical addresses. Even though a processor cannot directly reference another processor's memory, it can contain pointers (virtual IDs) to objects there and can request actions by sending messages to the processor containing an object. Each processor manages its own translation between virtual IDs and physical memory addresses; it can, for example, relocate objects without having to notify any other processors. The translation is flexible enough to allow an object to easily migrate from one node to another or even to a disk.

Other Communication Mechanisms

In addition to efficient message sending and receiving, there are other useful message-passing communications mechanisms. One useful feature which neither Active Messages nor the J-Machine provides is atomicity of message sends. This permits a processor to cancel a message send if the entire message cannot be accepted by the network and is very useful in avoiding deadlocks in protocols; for example, the distributed cache coherence scheme in [43] optimizes out many network operations. Due to dependency cycles (Figure 11), this would not be possible if processors were committed to sending the rest of the message once they initiated a message send⁹.

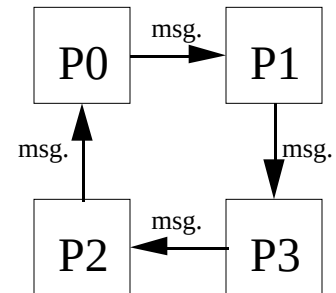


Figure 11. Message Deadlock

If one or more processors try to send messages to each other in a cycle and the network buffers fill up while the processors all wait for the network, the machine will deadlock. Deadlocks can be avoided by canceling message sends, moving data from the network buffers to some other place, or by using multiple priority levels or virtual channels so cycles don't exist.

In many applications the availability of several priority levels allows simple calls/returns without the need to allocate large buffers or worrying about deadlocks. Moreover, separate priority levels are useful for the supervisor and for debugging purposes.

Another set of mechanisms concerns protection. If a message-passing computer is time- or space-shared, the processes should not interfere with one another. This includes network contention—a process should not be able to saturate another's network. This goal can be accomplished by having the processes run in disjoint parts of the machine with hardware message destination address checking or by having them use different virtual network channels¹⁰. Virtual network channels can guarantee minimum bandwidths to processes without wasting wire bandwidth if not all channels are busy [8]. Another element of protection is the ability to drain a portion of a network of messages without dispatching them for debugging purposes or to time-share processes. The CM-5 accomplishes this with an "all-fall-down" mode where messages currently in the network are dropped into nearby processors; this has the advantage of requiring only fixed-size buffers on the processors.

⁸This isn't done now due to quirks in the instruction set.

⁹The J-Machine handles this problem by transferring messages into an overflow buffer if a message queue overflows; if the overflow buffer also overflows, a higher-priority network level must be used and things get quite complicated.

¹⁰A virtual network channel appears to be a logically separate network channel but actually shares wires with other network channels.

iWarp [8] provides an additional facility for systolic communication. A pair of nodes can establish a channel between them for systolic communication, thereby reserving a virtual network link between the two nodes. The two nodes are then able to communicate individual words quickly, without routing information. This facility permits non-blocking emulation of various network topologies and may be useful for multimedia applications which require a guaranteed bandwidth.

Multitasking

Message-passing mechanisms are closely related to multitasking. It would be wasteful (and cause deadlocks) to make a processor wait for a result from a remote function call before proceeding; a superior solution is to switch to another task while the original one is waiting for a remote reply. The speed of context switches greatly affects the programming style of the machine—slow context switches encourage the use of coarse-grain parallelism and busy-waiting.

On conventional processors, even Active Messages on the CM-5, context switches require a sizable amount of time to switch the process state. J-Machine context switches take about 30 instructions, which is still large. Machines exist such as HEP [30] and Monsoon [39] that have zero context switching times (as long as the set of active contexts doesn't over- or underflow), but they suffer from poor sequential performance. The 88110-based dataflow architecture *T [37] has faster context switching at the expense of reducing the register set available to sequential tasks and imposing constraints on the compiler. Ideally, a machine should have near-zero context switching time while also having good sequential performance; the M-Machine tries to achieve that goal.

Unfortunately, fine-grain parallelism can negate some of the benefits of local caches because the working set becomes larger. TAM [13] attempts to group executions of related tasks together, but this area is still relatively unexplored.

4. Distributed System Communication

Unlike shared memory and message passing, communication in distributed systems is much less homogeneous. There are a number of applications and systems for communication, with no particular abstraction being dominant except at the lowest OSI levels [50] such as unreliable packet delivery at the network layer (see Figure 12). Some systems provide only file system services, others allow remote procedure calls, while still others provide higher-level application services. Continuing with the direction of the previous sections, this section will concentrate on services that would be useful within an

application running on several processors or computers. Such services tend to be function libraries and higher-level operating system services rather than mere distributed filing systems.

Several approaches in research systems are described below. There are many others; some, such as using AppleEvents or OCE will be familiar to many readers.

Layer	Services provided
Application	User services
Presentation	Exchange of structured data
Session	Establish and maintain session bindings between communicating entities
Transport	Transport data between session entities
Network	Route packets over network
Datalink	Send packets over a single physical network segment
Physical	Exchange bits

Figure 12. OSI Reference Model

A network organization conforming to the OSI Reference Model is composed of distinct abstractions built on top of one another, each one implementing new functionality on the one below it. OSI systems provide a wide set of functionality at the expense of high overhead needed to interpret higher-level protocols in terms of lower-level ones.

Systems

PVM [41][19] is an example of a communication architecture for a distributed system of heterogeneous computers. PVM is a coarse-grain protocol for communicating among programs that implements data translation, shared memory emulation, process and event operations, broadcasts, barriers, and locks. Machines with different processors or architectures can transparently communicate with each other; data formats are automatically translated and the appropriate program binaries selected. A PVM parallel programs can run simultaneously on a vector supercomputer, highly parallel machine, and a network of workstations, taking advantage of each architecture's best features. PVM is implemented as a library and supports a variety of source languages. One

nice touch in PVM's is an X-Windows graphical interface, HeNCE, for viewing and debugging parallel programs [18]

PVM assumes only that unreliable, unsequenced, point-to-point communication facilities exist among the participating computers; a test implementation of PVM uses the UDP protocol. Shared memory, barriers, and broadcasts are emulated with point-to-point communications, making them inefficient—an 8-byte message takes 15ms, and a 64-process barrier takes 0.6 seconds! Clearly, an application must be divided into large chunks in order not to be swamped by communication overhead. PVM is most useful for interfacing large modules to each other such as orchestrating communication between a vector supercomputer, a parallel computer, and a graphics workstation for visualizing the data. It can also be used to take advantage of idle time on networks of workstations as long as the problem does not require low-overhead communication.

The Nectar system [6] has the same goal as PVM—computation on a network of heterogeneous computers—but it uses lower-level facilities to overcome PVM's speed problems caused by traversing the OSI hierarchy. In particular, Nectar uses a custom high-speed fiber-optic network for high-throughput, low-latency communication among the nodes. In the spirit of iWarp, both circuit and packet switching are supported, and communication paths can be programmed; in addition, a communication path can be split in the network to implement multicasting. Nectar uses plug-in cards that implement memory-mapped, user-level I/O to eliminate the overhead of kernel calls for communication while retaining compatibility with existing computers. RISC processors called CABs handle communication activities for the host nodes.

Emerald [27][31] is a complete language and environment for programming distributed systems. Unlike many other parallel and distributed languages, Emerald provides a single model for sequential and parallel computing: programs work with the same kinds of objects both locally and globally. On the other hand, Emerald is running on distributed clusters of workstations where local computation performance is orders of magnitude greater than network latency, so there is considerable incentive for the system to classify object usages as local or global in order to permit optimizations; the system can do this automatically in some cases. In order to improve efficiency, Emerald can move parameter objects to the site of a remote function call at the time the call is made. Emerald communication is coarse-grained and requires overhead to translate object addresses between the nodes' local address spaces. This permits fast sequential computation at the expense of extra computation on message sends.

Amber [12] is an implementation, based on Emerald, of C++ on a distributed network of homogeneous workstations, each of which contains several processors communicating by shared memory. Unlike Emerald, Amber maintains a global address space for all objects in the system and simply maps

them in and out of local memories using virtual memory techniques; this helps reduce Amber's grain size. Unless specified otherwise, when a thread references a remote object, the thread will travel to the object rather than the object migrating to the thread's current location.

Linda [4] is a communications kernel that can be interfaced to many languages. It defines a global tuple database with three basic operations: adding, removing, and reading tuples. Tuples can be selected by matching on arbitrary elements. Implementations of Linda exist for various shared-memory and message-passing systems, although the message-passing implementations in [4] did not appear to be particularly efficient or scalable—they either replicated the tuple space over every processor or broadcast match requests to every processor.

Multiple Representations

When concurrent programs are run on a network of heterogeneous machines, problems of differing representations of data and code often arise. Problems appears both at the low level of byte ordering and at higher levels of data structures and content. The traditional approach is to define a canonical representation for communicated data such as XDR [36] and to produce different code images for different machines as is done in PVM. Unfortunately, if the canonical data representation is different from either the source or destination representation, two conversions will result instead of one or even none. One solution is to negotiate a common representation that is efficient for at least one of the parties to the communication. More involved protocols can also be used [45], with extraordinary gains in cases where converting data to and from the canonical form is slow. Unfortunately, this approach does not work as well if the communication is deferred over time.

Programs designed for a parallel computer can be run on heterogeneous machines by compiling modules for the different machines. However, migration becomes difficult, as the state of a running

process is difficult to capture in a machine-independent way. Incredibly, [44] provides an approach based on incremental recompilation¹¹, but it restricts compiler optimizations to require the machine state to agree with the source program on procedure calls.

Robustness

Atomicity, reliability, and security play a very important role in communications. Some systems, especially long-lived or distributed ones, can recover from communication errors or even node crashes. Argus [35] is the classic robust system, with support for nested atomic transactions and fault recovery. Although robustness is a system and language-design issue, decisions about reliability have to be visible at the communication abstraction level as well; they should not be added as an afterthought [7].

5. Discussion

Is there an Ideal Substrate?

One goal of research in communication abstractions is to find an abstraction that is usable on a wide range of systems for a wide range of applications. An example of such an abstraction is the use of virtual memory on sequential machines for managing the memory hierarchy—the vast majority of programs can disregard virtual memory details and run just fine¹².

Cache-coherent shared memory is one candidate for being a nearly universally applicable abstraction for parallel systems of today's technology—it is fast, scalable (if directory schemes are used), and nearly all paradigms can be implemented efficiently with it. Using shared memory does involve hardware costs, but they are small compared to the costs of writing software.

Active Messages is another candidate for a universal abstraction—it too is fast and scalable. Current implementations still have too much overhead to allow efficient emulation of the shared memory paradigm using messages, but this will likely change.

Message-passing can be implemented on top of shared memory, and shared memory can be implemented on top of message passing, a concept recognized in Mach [49]. In the end, cache-coherent shared memory and Active Messages will merge into one paradigm that supports memory requests as a special kind of messages, handled mostly in hardware. Systems may differ in the amount of hardware assistance they provide for memory request messages, but this appears

to be the most flexible combination for compact systems. The common cases can be handled less expensively in hardware, while many of the esoteric cases can be done by software. Nevertheless, there are several issues which have to be addressed before this happens, as outlined below.

Local Needs

Future processor will have to include support for efficient task switching to mask the hundreds of instruction opportunities that will be wasted by a memory fetch caused by a cache miss; the same mechanism will help keep the processor busy while it is waiting for network memory references and remote procedure call results. Low-latency communications also require fast response time from the target of a message. Current RISC processors have remarkably poor interrupt response times due to the need to switch large register files and process states. At the same time, processor designers are running out of ideas to take advantage of the parallelism inherent in a stream of instructions; four or five instructions at a time seems to be the maximum [46]. Both of these problems can be neatly solved by building a multithreading processor or a chip with several independent processing units sharing memory; some threads perform computation, while others serve remote requests. iWarp and *T are following this approach.

Protection

The mechanism of address space mapping and inter-process protection will also change. A large reason for the high overhead of process switches on current computers is the need to remap address spaces to enforce protection. This is impractical on processors that perform a context switch every nanosecond, so a different mechanism has to be used. Fortunately, such a mechanism exists and has been known for a long time—capability-based protection.

¹¹Can you imagine running a program on a Mac, going into the debugger, saving its state, and then resuming it under Microsoft Windows on an IBM clone, where in both cases the program runs at full speed in native mode?

¹²However, the situation is now changing with greater use of interactive and real-time applications on workstations; unpredictable delays can be quite bothersome when running multimedia applications.

In a simple form of capability-based protection, memory is divided into variable-size segments; an address consists of a segment descriptor and an offset (this need not bring up the 80x86 nightmares—segments can be arbitrarily long or short and are completely orthogonal to the paging mechanism). Offsets are integers, but segment descriptors are words marked with a special bit that can only be changed by the kernel. A user program can read, write, copy, or compare for equality segment descriptors in registers and memory at will, but it cannot forge or change them. Thus, a user program can only access memory for which it has segment descriptors, either generated by operating system calls or because it found one somewhere in memory it can reference. Programs can share segments simply by passing segment descriptors to each other. Under capability-based protection, user programs are completely unaware of where in the address space they reside, and there is no need to reconfigure the MMU between processes.

Other Issues

Some applications such as real-time video manipulation can benefit from bandwidth guarantees provided by circuit switching. Bandwidth is hard to guarantee using plain message passing, so for these applications the network channels should be software-configurable at a lower level as in iWarp and Nectar. Bandwidth guarantees or priority levels are also useful for security reasons to prevent a runaway user program from starving out the operating system.

Other applications such as timesharing and garbage collection require the ability to inspect the messages currently in the network and optionally remove them from the network. This can be done as in the CM-5 or by providing a mode where messages are not executed by receiving nodes.

Network hardware reliability is a thorny issue. The current practice is to use software to provide robust services on distributed networks and to ignore the problem (or merely detect faults and halt the machine if one occurs) on parallel computers. A software solution is undesirable here because it requires the originating node to keep track of messages it sends out until it knows that they are received, thus involving extra copying and overhead. The efficiency and simplicity goals of Active Messages would be violated unless the communications substrate can support reliable message routing and delivery. Of course, the difficult problem of what to do when a node crashes remains.

Distributed Needs

The needs of distributed networks overlap those of parallel computers, but they also extend over a much broader range. Some localized distributed networks can act like parallel computers, but others require facilities such as wide-area routing and naming, extensive security, and tolerance of high latency and low bandwidth. Public network security considerations alone preclude efficient user-level network access of shared memory and Active Messages; both of

these systems rely on hardware source checking of destination addresses or capability-based protection, which doesn't work well on a public network, where anyone can snoop on wires, intercept messages, modify and misroute them, and flood networks with spurious data. Encryption schemes exist to reliably deal with all of these problems, but they are time-consuming (and, unfortunately, haven't been implemented widely).

Of course, shared-memory and Active Message implementations can be supported on distributed networks to provide compatibility but don't work as well there because of the above problems and unavoidable high latency—there is no way to get around speed of light limitations, and we will have to deal with compatibility with existing networks for a long time. Thus, applications themselves have to be tuned to work around the effects of latency. The future will bring networks of hundreds of millions of computers interconnected via networks with an extremely large range of bandwidths. Protocols such as ISDN and ATM for handling this are starting to emerge, but this field is still in its infancy.

6. Conclusion

Parallel processing will move into the mainstream much more rapidly when applications can be easily ported and interfaced between machines of various architectures. The lack of standardization of communication architectures is a major stumbling block to this goal, but a universal communication paradigm—low-overhead message passing with support for shared memory—for parallel computers appears within reach. The situation is worse in the distributed arena, where the requirements span many orders of magnitude of bandwidth and latency and security needs.

Bibliography

- [1] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. "Comparison of Hardware and Software Cache Coherence Schemes." *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991, pp. 298-308.
- [2] Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz, John Kubiawicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. *The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor*. MIT Laboratory for Computer Science Technical Report 454, June 1991.
- [3] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. "An Evaluation of Directory Schemes for Cache Coherence." *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988, pp. 280-289.
- [4] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. "Linda and Friends." *IEEE Computer*, 19:8, August 1986, pp. 26-34.
- [5] Marco Annaratone, Claude Pommerell, and Roland Rühl. "Interprocessor Communication Speed and Performance in Distributed-Memory Parallel Processors." *Proceedings of the 16th International Symposium on Computer Architecture*, June 1989, pp. 315-324.
- [6] Emmanuel A. Arnould, François J. Bitz, Eric C. Cooper, H. T. Kung, Robert D. Sansom, and Peter A. Steenkiste. "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers." *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 205-216.
- [7] Henri E. Bal. "Fault-Tolerant Parallel Programming in Argus." *Concurrency: Practice and Experience*, 4:1, February 1992, pp. 37-55.
- [8] Shekhar Borkar, Robert Cohn, George Cox, Thomas Gross, H. T. Kung, Monica Lam, Margie Levine, Brian Moore, Wire Moore, Craig Peterson, Jim Susman, Jim Sutton, John Urbanski, and Jon Webb. *Supporting Systolic and Memory Communication in iWarp*. CMU Computer Science Technical Report CS-90-145, September 1990.
- [9] Michael Carlton and Alvin Despain. "Aquarius Project." *IEEE Computer*, 23:6, June 1990, pp. 80-83.
- [10] Brian Case. "IBM Delivers First PowerPC Microprocessor." *Microprocessor Report*, 6:14, October 28, 1992, pp. 1, 6-10.
- [11] Lucien M. Censier and Paul Feautrier. "A New Solution to Coherence Problems in Multicache Systems." *IEEE Transactions on Computers*, C-27:12, December 1978, pp. 1112-1118.
- [12] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. *The Amber System: Parallel Programming on a Network of Multiprocessors*. University of Washington Department of Computer Science Technical Report 89-04-01, April 1989.
- [13] David E. Culler, Anurag Sah, Klaus Erik Schausser, Thorsten von Eicken, and John Wawrzynek. "Fine-Grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine." *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 164-175.
- [14] William J. Dally et al. "Architecture of a Message-Driven Processor." *Proceedings of the 14th International Symposium on Computer Architecture*, June 1987, pp. 189-196.
- [15] William J. Dally et al. "The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms." *IEEE Micro*, 12:2, April 1992, pp. 23-39.
- [16] Michel Dubois, Christoph Scheurich, and Faye Briggs. "Memory Access Buffering in Multiprocessors." *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 434-442.

- [17] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauer. "Active Messages: a Mechanism for Integrated Communication and Computation." *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 256-266.
- [18] G. A. Geist and V. S. Sunderam. "Network-Based Concurrent Computing on the PVM System." *Concurrency: Practice and Experience*, 4:4, June 1992, pp. 293-311.
- [19] G. A. Geist and V. S. Sunderam. "The PVM System: Supercomputer Level Concurrent Computation on a Heterogeneous Network of Workstations." *Proceedings of the 6th Distributed Memory Computing Conference*, May 1991, pp. 258-261.
- [20] James R. Goodman. "Using Cache Memory to Reduce Processor-Memory Traffic." *Proceedings of the 10th International Symposium on Computer Architecture*, June 1983, pp. 124-131.
- [21] James R. Goodman, Mary K. Vernon, and Philip J. Woest. "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors." *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 64-73.
- [22] James R. Goodman and Philip J. Woest. "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor." *Proceedings of the 15th International Symposium on Computer Architecture*, June 1988, pp. 422-431.
- [23] Hendrik A. Goosen and David R. Cheriton. "Predicting the Performance of Shared Multiprocessor Caches." *Cache and Interconnect Architectures in Multiprocessors*, Michel Dubois and Shreekanth S. Thakkar, ed. Kluwer Academic Publishers, 1990, pp. 153-164.
- [24] Erik Hagersten, Seif Haridi, and David H. D. Warren. "The Cache Coherence Protocol of the Data Diffusion Machine." *Cache and Interconnect Architectures in Multiprocessors*, Michel Dubois and Shreekanth S. Thakkar, ed. Kluwer Academic Publishers, 1990, pp. 165-188.
- [25] R. Hannemann. "Physical Technology for VLSI Systems." *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, 1986, pp. 48-53.
- [26] Waldemar Horwat. *Concurrent Smalltalk on the Message-Driven Processor*. MIT Artificial Intelligence Laboratory Technical Report 1321, September 1991.
- [27] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. University of Washington Technical Report 87-01-01, January 1987.
- [28] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. "Scalable Coherent Interface." *IEEE Computer*, 23:6, June 1990, pp. 74-77.
- [29] Kirk L. Johnson. "The Impact of Communication Locality on Large-Scale Multiprocessor Performance." *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 392-402.
- [30] Harry F. Jordan. "Performance Measurements on HEP-A Pipelined MIMD Computer." *Proceedings of the 10th International Symposium on Computer Architecture*, June 1983, pp. 207-212.
- [31] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. "Fine-Grained Mobility in the Emerald System." *ACM Transactions on Computer Science*, 6:1, February 1988, pp. 109-133.
- [32] Kendall Square Research, Usenet comp.parallel mailing, February 22, 1992.
- [33] James T. Kuehn and Burton J. Smith. "The Horizon Supercomputing System: Architecture and Software." *Proceedings of Supercomputing '88*, November 1988, pp. 28-34.
- [34] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. "The DASH Prototype: Implementation and Performance." *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 92-103.
- [35] Barbara Liskov and Robert Scheifler. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs." *ACM Transactions on Programming Languages and Systems*, 5:3, July 1983, pp. 381-404.

- [36] B. Lyon. *Sun External Data Representation Specification*. Sun Microsystems, Inc. Technical Report, 1984.
- [37] Rishiyur S. Nikhil, Gregory M. Papadopoulos, and Arvind. **T: A Killer Micro for a Brave New World*. MIT Computation Structures Group Memo 325, July 1991.
- [38] Susan Owicki and Anant Agarwal. "Evaluating the Performance of Software Cache Coherence." *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 230-242.
- [39] Gregory M. Papadopoulos and David E. Culler. "Monsoon: An Explicit Token-Store Architecture." *Proceedings of the 17th International Symposium on Computer Architecture*, June 1990, pp. 82-91.
- [40] Steven L. Scott, James R. Goodman, and Mary K Vernon. "Performance of the SCI Ring." *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 403-414.
- [41] V. S. Sunderam. "PVM: A Framework for Parallel Distributed Computing." *Concurrency: Practice and Experience*, 2:4, December 1990, pp. 315-339.
- [42] Shreekant Thakkar, Michel Dubois, Anthony T. Laundrie, and Gurindar S. Sohi. "Scalable Shared-Memory Multiprocessor Architectures." *IEEE Computer*, 23:6, June 1990, pp. 71-74.
- [43] Manu Thapar and Bruce Delagi. "Stanford Distributed-Directory Protocol." *IEEE Computer*, 23:6, June 1990, pp. 78-80.
- [44] Marvin M. Theimer and Barry Hayes. "Heterogeneous Process Migration by Recompile." *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991, pp. 18-25.
- [45] Earl DeWitt Waldin III. *Using Multiple Representations for Efficient Communication of Abstract Values*. MIT Laboratory for Computer Science Technical Report 553, September 1992.
- [46] David W. Wall. "Limits of Instruction-Level Parallelism." *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 176-188.
- [47] Wolf-Dietrich Weber and Anoop Gupta. "Analysis of Cache Invalidation Patterns in Multiprocessors." *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 243-256.
- [48] Andrew W. Wilson Jr. "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors." *Proceedings of the 14th International Symposium on Computer Architecture*, June 1987, pp. 244-252.
- [49] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System." *ACM SIGOPS Operating System Review*, 21:5, November 1987, pp. 63-76.
- [50] Hubert Zimmermann. "OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection." *IEEE Transactions on Communications*, 28:4, April 1980, pp. 425-432.